) NTAINING 3RIDGE-CONNECTED AND BICONNECTED COMPONENTS ON-LINE

Jeffery Westbrook
Robert E. Tarjan

CS-TR-228-89

August 1989

N00014-87-K-0467

Maintaining Bridge-Connected and Biconnected Components On-line†

Jeffery Westbrook

Department of Computer Science
Princeton University
Princeton, New Jersey 08544

Robert E. Tarjan

Department of Computer Science
Princeton University
Princeton, New Jersey 08544
and
AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

We consider the twin problems of maintaining the bridge-connected components and the biconnected components of a dynamic undirected graph. The allowed changes to the graph are vertex and edge insertions. We give an algorithm for each problem. With simple data structures, each algorithm runs in $O(n \log n + m)$ time, where n is the number of vertices and m is the number of operations. We develop a modified version of the dynamic trees of Sleator and Tarjan that is suitable for efficient recursive algorithms, and use it to reduce the running time of the algorithms for both problems to $O(m\alpha(m,n))$, where α is a functional inverse of Ackermann's function. This time bound is optimal. All of the algorithms use $O(n)$ space.

1. Introduction

Three natural and important equivalence relations on the constituents of an undirected graph $G=(V,E)$ are defined by its connected, bridge-connected, and biconnected components. Let $V_1, V_2, \cdots, V_k$ be the partition of V such that two vertices are in the same part of the partition if and only if there is a path connecting them. Let G_i be the subgraph of G consisting of V_i and the edges of E incident to the vertices of V_i. The subgraphs G_i, $1 \le i \le k$, form the connected components, abbreviated *components*, of G. Similarly, let $E_1, E_2, \cdots, E_k$ be the partition of E such that two edges are in the same

† Research partially supported by National Science Foundation Grant DCR-8605962 and Office of Naval Research Contract N00014-87-K-0467.

part of the partition if and only if they are contained in a common simple cycle. The biconnected components, or *blocks*, are the subgraphs of G formed by the edges of E_i and the vertices of V that are endpoints of these edges. A vertex appearing in more than one block is called an *articulation point*, and its removal disconnects G. There is either no block or one block containing any given pair of vertices. An edge contained in no cycle is in a block by itself. Such an edge is called a *bridge*, and its removal disconnects the graph. The bridge-connected components, or *bridge-blocks*, of G are the components of the graph formed by deleting all the bridges. The bridge-blocks partition V into equivalence classes such that two vertices are in the same class if and only if there is a (not necessarily simple) cycle of G containing both of them. Figure 1 shows an undirected graph along with its blocks and bridge-blocks.

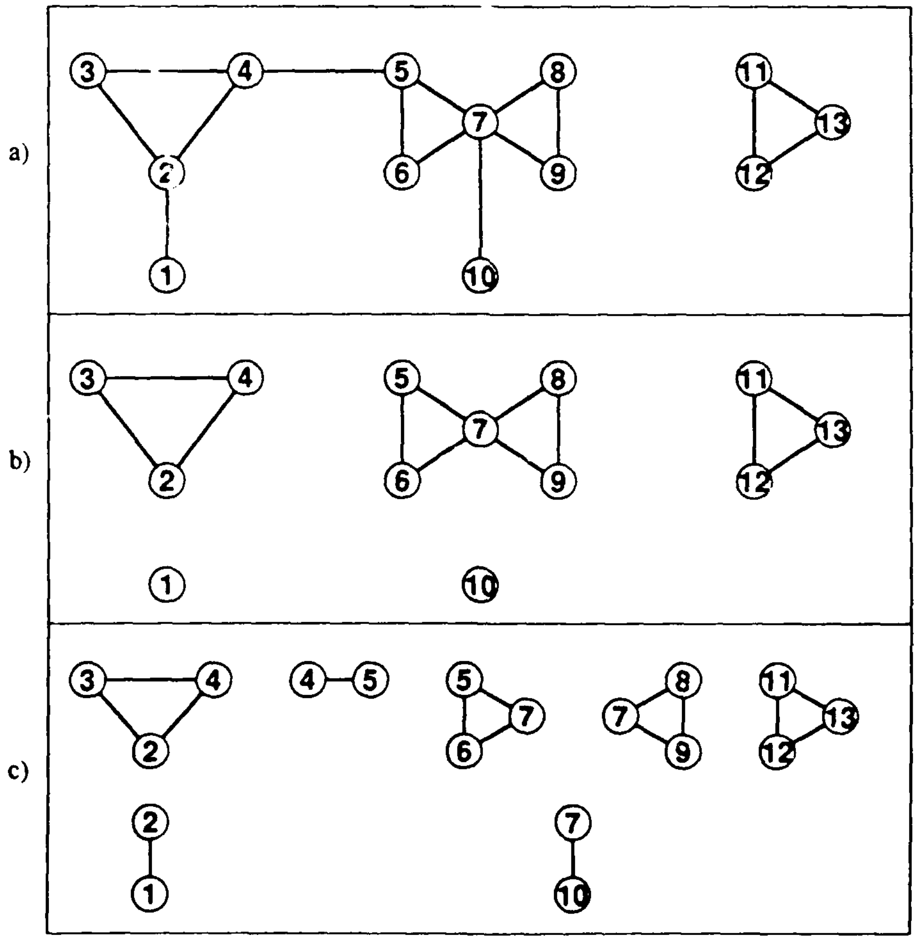


Figure 1: a) Undirected graph G. b) Bridge-blocks of G. c) Blocks of G. Multiply-appearing vertices are articulation points.

The problems of finding the components, blocks, and bridge-blocks of a fixed graph are well-understood. Hopcroft and Tarjan [9] and Tarjan [19] give sequential algorithms that run in time $O(n+m)$ where $n = |V|$ and $m = |E|$. Logarithmic-time parallel algorithms for finding components, bridge-blocks, and blocks are given in references [1] and [22] (see also the survey paper [11]).

The problem of answering queries about edge and vertex membership in the components of a dynamic graph, i.e., a graph that is changing on-line, has been addressed in references [3,5,7,12]. Even and Shiloach [5] consider the component problem for a graph undergoing edge deletions. They give algorithms with constant query time, $O(n \log n)$ total update time in the case that G is a tree or forest, and $O(mn)$ update time for general G, where m and n are the numbers of edges and vertices, respectively, in the initial graph. Reif [12] gives an algorithm for the same problem that runs in time $O(n\,g + n \log n)$ when given an initial graph embedded in a surface of genus g. Frederickson [7] gives an algorithm that performs queries in constant time and edge insertions and deletions in time $O(\sqrt{m_i})$, where m_i is the number of edges in G at the time of the i^{th} update. Even and Shiloach, and Reif, also observe that if only edge insertions are allowed, the component problem can be solved by straightforward application of a fast disjoint set union algorithm. The disjoint set union problem is to maintain a partition of n elements while performing an intermixed sequence of two operations: *find*(x), which returns the name of the set containing element x; and *union*(A,B), which combines the sets named A and B into a new set named A. The fastest algorithms for this problem run in $O(\alpha(m,n))$ amortized time per operation† and $O(n)$ space, where m is the length of the sequence, n is the total number of elements, and α is a functional inverse of Ackermann's function [16,20].

In this paper we study the problems of answering queries about the blocks or bridge-blocks of a dynamic graph. We allow two on-line graph update operations to be performed on an initially null graph G:

† The amortized cost of an operation is the cost of a worst-case sequence of operations divided by the number of operations in the sequence. See [17] for a general discussion of amortization.

make vertex (A): Add a new vertex with no incident edges to G. Label by "A" the bridge-block formed by the new vertex. Return the name of the new vertex; the calling program must use this name to refer to the new vertex in subsequent operations. (The name is actually a pointer into the data structure maintained by the algorithm.)

insert edge (u,v,A): Insert a new edge between the vertices named u and v. Label by "A" any new bridge-block or block that results from the edge insertion.

In the bridge-block problem we allow the following query:

find block (u): Return the label of the bridge-block containing the vertex named u.

Similarly, in the block problem we allow the following query:

find block (u,v): Return the label of the block, if any, containing the pair of vertices $\{u,v\}$.

We also consider a restricted variant of the problem in which we are given an initially connected graph $G_0 = (V_0, E_0)$. We allow $O(|E_0|)$ preprocessing time, and then process on-line a sequence of intermixed queries and edge insertions. In this variant, the *make vertex* operation is not allowed. Our algorithms do not explicitly maintain the edge set, but if the endpoints of an edge are known, then *find block* can be used to determine which block contains the edge.

The block and bridge-block problems are natural problems to consider in the general study of on-line graph algorithms, a study that has wide applications to networks, CAD/CAM, and distributed computing. They appear as subproblems of other on-line graph problems, such as incremental planarity testing [4]. The incremental planarity testing problem is to maintain a representation of a planar graph as edges are being added, and to determine the first edge addition that makes the graph nonplanar. Maintaining the blocks of a dynamic graph also arises in the implementation of efficient search strategies for logic programming [Graeme Port, private communication, 1988]. We know of no previous algorithms for the block or bridge-block problems that run in sublinear time per operation.

This paper is organized as follows. In Section 2 we develop a simple algorithm for the bridge-block problem that runs in $O(n \log n + m)$ total time and uses $O(n)$ space, where n is the number of vertices added to the initially null graph and m is the number of operations. In Section 3 we give a similar algorithm for the block problem that also runs in $O(n\log n + m)$ time and $O(n)$ space. In Sections 4-7 we decrease the total running time of the bridge-block and block algorithms to $O(m\alpha(m,n))$. To achieve this bound we introduce link/condense trees, a modified version of the dynamic trees of Sleator and Tarjan [14,15], and apply a fairly delicate analysis. The link/condense trees support condensing of adjacent nodes, and two such trees can be linked together in amortized time $O(\log k)$, where k is the size of the smaller tree. Section 4 describes the data structure as it applies to the bridge-block problem and Section 5 contains the amortized analysis of the link/condense tree operations. Section 6 describes the modifications needed to apply link/condense trees to the block problem, and Section 7 contains additional analysis. Finally, Section 8 contains a simple reduction from the disjoint set union problem to the block and bridge-block problems. This allows us to apply the known lower bounds for set union to these two problems. Our results are summarized in the following table:

	Initially connected G	Initially null G
Bridge-blocks	$O(m\alpha(m,n))$	$\Theta(m\alpha(m,n))$
Blocks	$\Theta(m\alpha(m,n))$	$\Theta(m\alpha(m,n))$

The $O(\alpha(m,n))$ upper bound for the on-line bridge-block and block problems is somewhat surprising, since both these problems differ fundamentally from the on-line connected component problem that shares this bound. In the latter, a single edge insertion can combine at most two components into one. A single edge insertion, however, can create a cycle in the graph that might combine as many as $\Theta(n)$ blocks or bridge-blocks into one. This fact seems to make the maintenance of blocks and bridge-blocks under both edge insertion and edge deletion quite difficult. By simply alternating edge insertions with edge deletions, we can generate a sequence of operations that at every step changes the number of blocks in the graph by $\Theta(n)$.

To conclude this section, we note that although *find block* as defined above returns only a label, the algorithms presented below can be extended to return other information about the blocks or bridge-blocks, such as an edge or vertex of maximum weight, with no loss in efficiency. In general, we can maintain any data that can be updated in constant time when two blocks or bridge-blocks are combined into one. If we are willing to increase the space used to $O(m)$, we can also list the edges or vertices in a block or bridge-block in time $O(\alpha(m,n)+k)$, where k is the number of items listed.

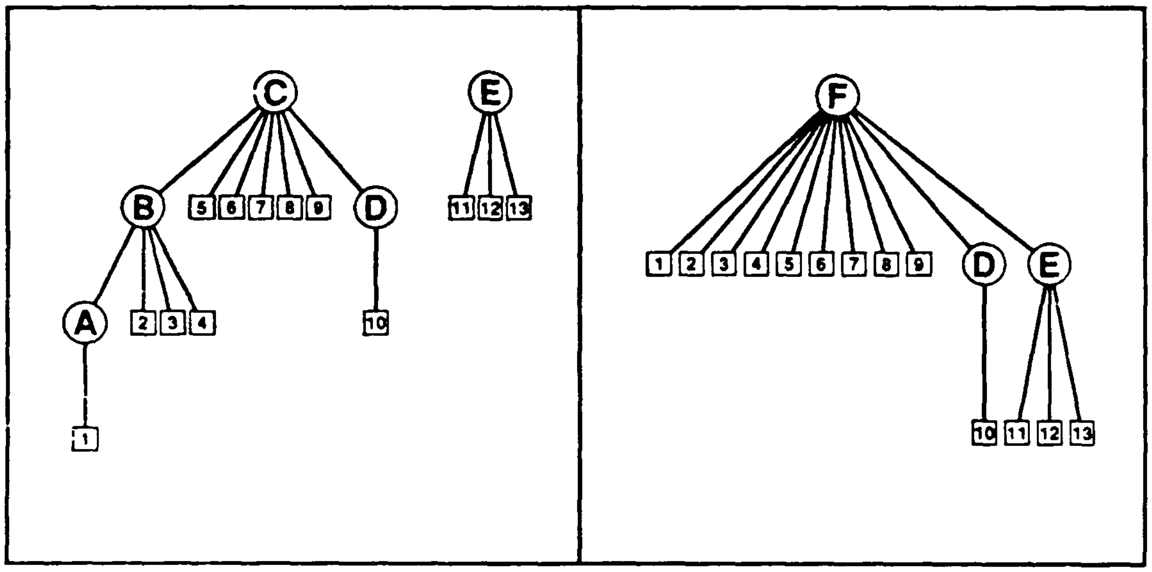
2. Maintaining Bridge-Blocks On-Line

The bridge-blocks and bridges of a connected graph have a natural tree structure that we call the *bridge-block tree*. The collection of bridge-block trees given by the components of a graph $G=(V,E)$ is called the *bridge-block forest*, or BBF. The nodes in a bridge-block tree are of two types: square nodes, which represent the vertices of G; and round nodes, which represent the bridge-blocks. If r is a round node then $label(r)$ denotes the label of the corresponding bridge-block. Two round nodes are connected by a tree edge whenever a bridge connects the corresponding two bridge-blocks. If vertex v is contained in bridge-block A then the square node that represents v is connected by a tree edge to the round node that represents A. Every square node is a leaf. Each bridge-block tree is rooted at an arbitrarily selected round node. We denote by $parent(x)$ the parent of tree node x. Given the BBF for G, the query *find block* (u) can be answered by computing $label(parent(u))$. (In general, we let the vertex name, here u, refer to both the vertex in the graph and to its square node representative in the bridge-block tree.) Figure 2a shows the BBF for the graph of Figure 1.

The effect of an *insert edge* (u,v,A) operation on the bridge-block forest depends on whether u and v are in the same bridge-block, different bridge-blocks but the same component, or different components. In the first case the bridge-block structure of the graph, and consequently the BBF, is unaffected. In the remaining two cases, however, the bridge-block structure is affected in opposite ways.

If u and v are in different components of the graph then the inserted edge connects the two components by creating a bridge between them. One of the two corresponding bridge-block trees, say the tree containing u, is rerooted at $parent(u)$, and then $parent(u)$ is made a child of $parent(v)$. This process, called a *component link*, can occur at most $n-1$ times, after which the graph is connected.

On the other hand, if u and v are in the same component but different bridge-blocks, the inserted edge creates a new cycle that reduces the number of bridge-blocks in the component. The round nodes on the path connecting u and v in the corresponding tree must be replaced by a single round node. Every node previously adjacent to one of the round nodes on the path becomes adjacent to the new single round node. This process is called *path condensation*. After at most $n-1$ condensations, the graph has only a single bridge-block. Figure 2b gives examples of both cases of edge insertion.



(a) (b)

Figure 2: a) Bridge-block forest of G. b) BBF after performing *insert edge* $(1,5,F)$ and *insert edge* $(9,11,E)$

To implement the bridge-block operations, the BBF is explicitly maintained with a data structure that supports the following functions.

Maketree (A : label or null value):
    Create a new tree consisting of a single node. If A is null, make a square node; otherwise, make a round node labeled A; Return a pointer to the new node.

Link (x,y :square or round nodes):
    Link two trees together by first making x the root of its tree (this is called an *eversion*), and then making x a child of y.

Findpath (u,v : square nodes):
    Find the tree path P between but not including square nodes u and v. If u and v are the same node, return *parent* (u).

Condensepath (P : *path* ; A : label):
    Perform path condensation on P and label by A the the resultant single node $\tilde{r}$.

With these functions, we implement the bridge-block operations as follows:

make vertex (A):

Let $u = \mathit{Maketree}(\text{null})$. Perform $\mathit{Link}(u, \mathit{Maketree}(A))$. Return u (to be used by the calling program as the name of the new vertex).

find block (u):

Return $\mathit{label}(\mathit{Findpath}(u,u))$.

insert edge (u,v,A):

(1) If u and v are in the same component then execute $\mathit{Condensepath}(\mathit{Findpath}(u,v),A)$ and terminate.

(2) If u and v are in different components, determine which is in the smaller component. Assume u is. Let $x = \mathit{Findpath}(u,u)$ and $y = \mathit{Findpath}(v,v)$.

(3) Execute $\mathit{Link}(x,y)$.

In the *insert edge* procedure we use an on-line component maintenance subroutine to determine if two vertices are in the same component of G and to determine the size of a component. This subroutine is a straightforward application of a fast disjoint set union algorithm [21]. Appropriate calls to the update functions of this subroutine must be made when making a new vertex or performing a component link.

The tree data structure is built using *condensible nodes*. A condensible node x consists of a block of storage, $N(x)$, containing an arbitrary but fixed collection of fields, and a set of subnodes, $S(x)$. The subnode sets are maintained with a fast disjoint set union algorithm [21]. The name of set $S(x)$ is simply $N(x)$. A condensible node is initialized with one subnode. To make a pointer p to node x, we store in p the name of some subnode $s \in S(x)$. Given such a pointer p, a *pointer step* consists of accessing $N(x)$ by performing $\mathit{find}(p)$. Two nodes x and y can be condensed into a single node z by the following procedure: create a new storage block $N(z)$; let $S(z) = \mathit{union}(S(x),S(y))$; update appropriately the fields of $N(z)$ using the fields of $N(x)$ and $N(y)$; and discard $N(x)$ and $N(y)$. The union of the two subnode sets suffices to make all pointers to x and y become pointers to z. Note that condensation destroys x and y. If a data structure initially contains n condensible nodes, then any sequence of m pointer steps and condensations runs in worst-case time $O(m\alpha(m,n))$, and the data structure uses $O(n)$ space.

For each vertex v in the graph we store a pointer to its condensible node representative in the data structure. For a node x, N(x) contains a parent pointer and label field. If x is a square node, the label field is null; if x is a round node, the label field holds a bridge-block label. For any node x, $parent(x)$ is computed by a pointer step using the parent pointer. The four tree functions take as arguments pointers to condensible nodes.

To perform $Link(x,y)$, the tree containing x is everted by walking up from x to the tree root, reversing the direction of all parent pointers along the path. The number of pointer steps is the initial depth of x, which is at most the size of the tree. After the evert, a pointer to y is stored in the parent field of x. Since x and y are passed in the form of pointers, this is actually implemented by storing the value of y in the parent field of the storage block returned by $find(x)$. This requires one additional pointer step.

The $Findpath(u,v)$ algorithm proceeds by walking up the tree simultaneously from u and from v in lock-step, until the paths from u and v intersect at their nearest common ancestor. The path P is returned as a list of nodes (not in order along the path), with the nearest common ancestor at the end. The number of pointer steps required is at most $2|P|$. Let z be the parent of the nearest common ancestor. To perform $Condensepath(P,A)$, P is condensed into a single node $\tilde{r}$. Node $\tilde{r}$ is labeled by A and made a child of z. The number of node condensations is $|P| - 1$.

Lemma 1:

In any sequence of operations there are $O(n)$ pointer steps during condensing edge insertions.

Proof. Suppose a path P is being condensed. To generate P, $O(|P|)$ pointer steps are required, but $|P| - 1$ round nodes are condensed. After $n-1$ condensations the graph is bridge-connected. □

Lemma 2:

The total number of pointer steps during Link operations is $O(n\log n)$.

Proof. The number of square nodes in a bridge-block tree is at least the number of round nodes. Hence the cost of an evert in a tree of k square nodes is $\Theta(k)$. Since we always evert the smaller tree, we arrive at the following recursive upper bound on the total number of pointer steps, $T(n)$, needed to combine n components into one, where c is a sufficiently large constant.

$$T(1) = 0$$

$$T(n) \le \max_{1 \le k \le n/2} \{T(k) + T(n-k) + ck\}$$

It is well-known that this recurrence has the solution $T(n) = O(n \log n)$. □

Theorem 1:

Given an initially null graph G_0, a sequence of $m = \Omega(n)$ *find block, make vertex,* and *insert edge* operations can be processed in $O(n \log n + m)$ time and $O(n)$ space, where n is the number of vertices inserted.

Proof. First, we note that the total time spent in the on-line components subroutine is $O(m\alpha(m,n)) = O(n \log n + m)$. Next, we bound the total number of pointer steps and condensations that occur during processing of an input sequence. There is at least one pointer step per bridge-block operation. Let k be the number of pointer steps and node condensations occurring during path condensation and component linking. From lemmas 1 and 2, $k = O(n \log n)$. The amortized cost of a pointer step or node condensation is $O(\alpha(k+m,n))$. Thus the total running time is $O((k+m)\alpha(k+m,n))$. Since $\alpha(m,n) = 1$ for $m \ge n \log\log n$ [18], this expression is $O(n \log n + m)$. The space bound follows from the properties of condensible nodes and the observation that the number of square nodes bounds the number of round nodes. □

Corollary:

Given an initially connected graph G_0 and $O(|E_0|)$ preprocessing time, a sequence of $m = \Omega(n)$ *find block* and *insert edge* operations can be processed in $O(m\alpha(m,n))$ time.

Proof. The bridge-blocks and initial BBF of G_0 can be found in time $O(|E_0|)$ using one of the algorithms in references [9,19]. By lemma 1, the total number of pointer steps and condensations is $O(m)$, giving the bound. □

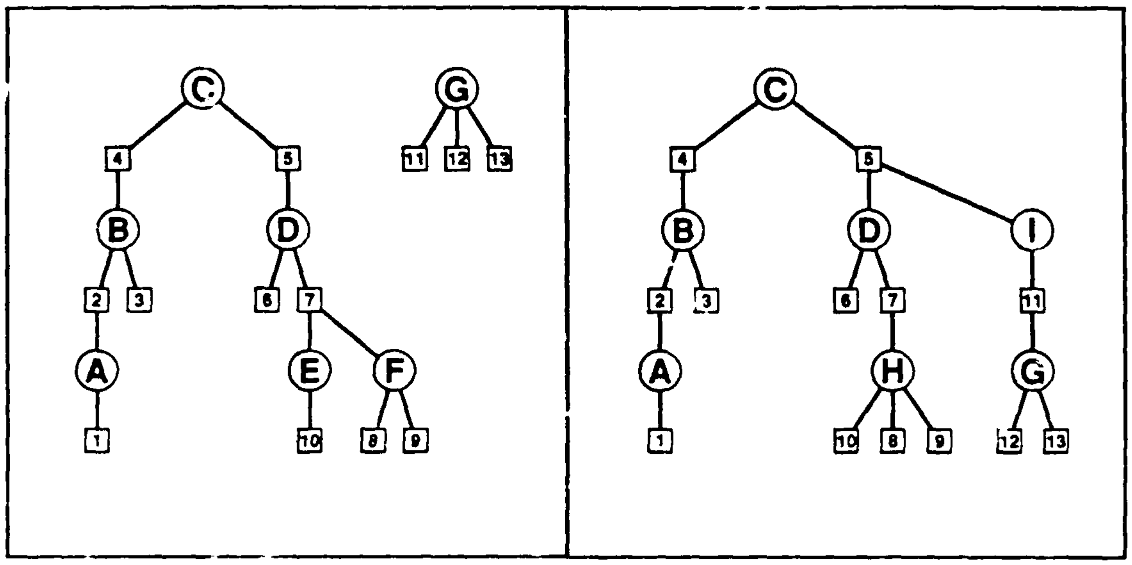
3. Maintaining Blocks On-line

The problem of maintaining blocks on-line is similar to that of maintaining bridge-blocks, and the algorithms are almost identical. We represent each block of G by a round node and each vertex by a square node. The square nodes now play a more important role, however, and the *block forest*, or BF, is different in character from the bridge-block forest. Whenever a vertex of G belongs to a given block, we create a tree edge between the corresponding square and round nodes in the BF. There are no other edges; no two square nodes and no two round nodes are adjacent. Since a single vertex can be an articulation point joining many blocks, and a block may be adjacent to many articulation points, the block tree generally has internal square nodes. If $\{u,v\}$ is a vertex pair that appears in block B, then in the block tree either u and v are both children of B or one is a parent of B and one is a child of B. The query *find block*(u,v) can be answered by returning the label of the single round node that lies on the tree path between u and v. Figure 3a gives the block forest for the graph of Figure 1.

A *make vertex*(u) operation adds a single unconnected square node to the block forest. An *insert edge*(u,v,A) operation can have two opposite effects: either it links two components, increasing the number of blocks, or it creates a new cycle, possibly decreasing the number of blocks. If a component link occurs, then the inserted bridge forms a new block labeled by A. One of the two block trees, say the tree containing u, is rerooted at u; u becomes a child of A, and A becomes a child of v.

If the inserted edge creates a new cycle then path condensation occurs. In a block tree the path between u and v consists of alternating square and round nodes. The round nodes on the path are condensed into a single round node, $\tilde{r}$. The square nodes on the path may be ignored, since condensation of the round nodes ensures that these square nodes become children of $\tilde{r}$. Let w be the nearest common ancestor of the path nodes. (Recall that this implies that w is part of the path.) If w is a round node, then $\tilde{r}$ becomes a child of the parent of w. If w is a square node, it is not affected by condensation, and $\tilde{r}$ becomes a child of w. Examples of a linking edge insertion and a condensing edge insertion with a square nearest common ancestor are given in Figure 3b.

We maintain the block forest with the same condensible node data structure used for the bridge-block problem. The *Link, Findpath,* and *Condensepath* functions described in the previous section can be modified in a straightforward way to perform the block tree transformations described above. With these functions, the block operations are implemented as follows:



(a) (b)

Figure 3: a) Block forest of G. b) BF after performing *insert edge* $(8,10,H)$ and *insert edge* $(5,11,I)$.

make vertex (A):

Return *Maketree* (null). (This does not create a block, so label A ignored.)

find block (u,v):

Return *label* (*Findpath* (u,v)).

insert edge (u,v,A):

(1) If u and v are in the same component then execute *Condensepath* (*Findpath* (u,v),A) and terminate.

(2) If u and v are in different components, determine which is in the smaller component. Assume u is.

(3) Let $\hat{r}$ = *Maketree* (A).

(4) Execute *Link* $(u,\hat{r})$. Execute *Link* $(\hat{r},v)$.

The lemmas and analysis given for the bridge-block problem can be easily adapted to the block problem.

Theorem 2:

The on-line block problem can be solved in $O(n\log n+m)$ time and $O(n)$ space.

Corollary:

If the graph is initially connected, the on-line block problem can be solved in $O(m\alpha(m,n))$ time.

4. A Data Structure for an Optimal Bridge-Block Algorithm

In this section we replace the data structure used in section 2 with a more sophisticated data structure called the *link/condense tree*. We observe that in the bridge-block algorithm of section 2 most of the work occurs in processing component links. The link/condense tree is designed to perform everts efficiently, and hence speed component links, without increasing the time to perform *Findpath* and *Condensepath*. By maintaining the bridge-block forest with link/condense trees, any sequence of $n-1$ component links can be performed in $O(n\alpha(m,n))$ time, and it remains the case that $n-1$ path condensations take time $O(n\alpha(m,n))$.

The link/condense tree is derived from the dynamic tree data structure of Sleator and Tarjan [14,15]. For a full description of dynamic trees the reader should consult their papers. Below we will consider mainly the new aspects of link/condense trees. The following summary description of dynamic trees is taken from [15, p. 678]. (See Figure 4.)

> We represent each dynamic tree T by a *virtual tree* V containing the same nodes as T but having a different structure. Each node of V has a left child and a right child, either or both of which may be a null, and zero or more middle children. We call an edge joining a middle child to its parent *dashed* and all other edges *solid*. Thus the virtual tree consists of a hierarchy of binary trees, which we call *solid subtrees*, interconnected by dashed edges. The relationship between T and V is that the parent in T of a node v is the symmetric-order successor of v in its solid subtree in V, unless v is last in its solid subtree, in which case its parent in T is the parent of the root of its solid subtree in V. In other words, each solid subtree in V corresponds to a path in T, with symmetric order in the solid subtree corresponding to linear order along the path, from first vertex to last vertex.

The link/condense virtual tree is built with condensible nodes. Let v be a tree node contained in solid subtree U. $N(v)$ contains pointers to the following nodes: *vparent*(v), the parent of v in the virtual tree V; *left*(v) and *right*(v), the left and right children of v in the solid subtree; and *pred*(v) and *succ*(v), the symmetric order predecessor and successor of v in the solid subtree. In general, *succ*(v) points to the parent of v in T. In addition, $N(v)$ contains pointers *leftmost*(v) and *rightmost*(v). These are used only when v is the root of a solid subtree, at which time they point to the leftmost and rightmost nodes in this solid subtree. If t is the root of U and $l = \mathit{leftmost}(t)$, $r = \mathit{rightmost}(t)$, then $\mathit{pred}(l) = \mathit{succ}(r) = t$.

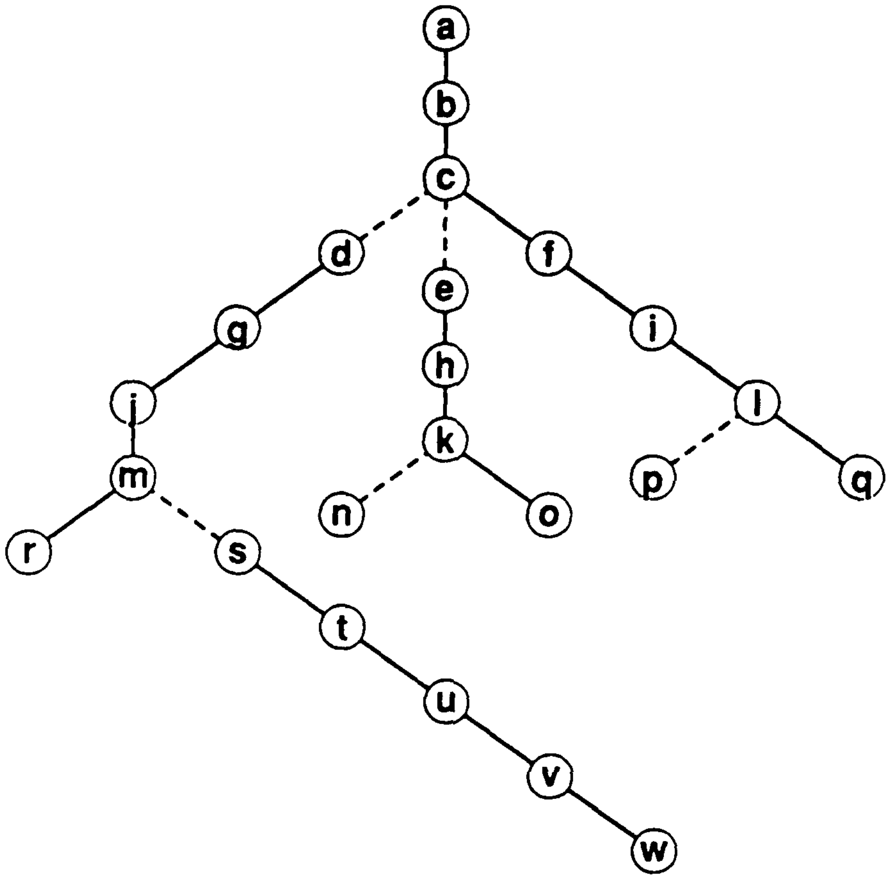


Figure 4a: A dynamic tree: the actual tree. Dashed edges separate paths corresponding to solid subtrees in the virtual tree [15].

As a way to implement eversion efficiently, we allow a node v to occasionally enter *reversed state*, in which case the meanings of *left*(v) and *right*(v), *leftmost*(v) and *rightmost*(v), and *pred*(v) and *succ*(v) are reversed. (That is, *left*(v) points to the right child, *right*(v) points to the left child, etcetera.) To implement reversal, $N(v)$ contains a bit *reverse*(v). The reversal state of node v is given by the exclusive-or of the *reverse* values stored in v and all its ancestors in the solid subtree.

The solid subtrees are built using splay trees [15]. A *splay at node x* moves x to the root of its solid subtree by applying a standard binary tree rotation to every edge along the path from x to the root (Figure 5a). A rotation rearranges left and right children while preserving the symmetric order. Middle children are unaffected. The sequence of rotations is determined by the structure of the path [15]. To perform links efficiently, we use

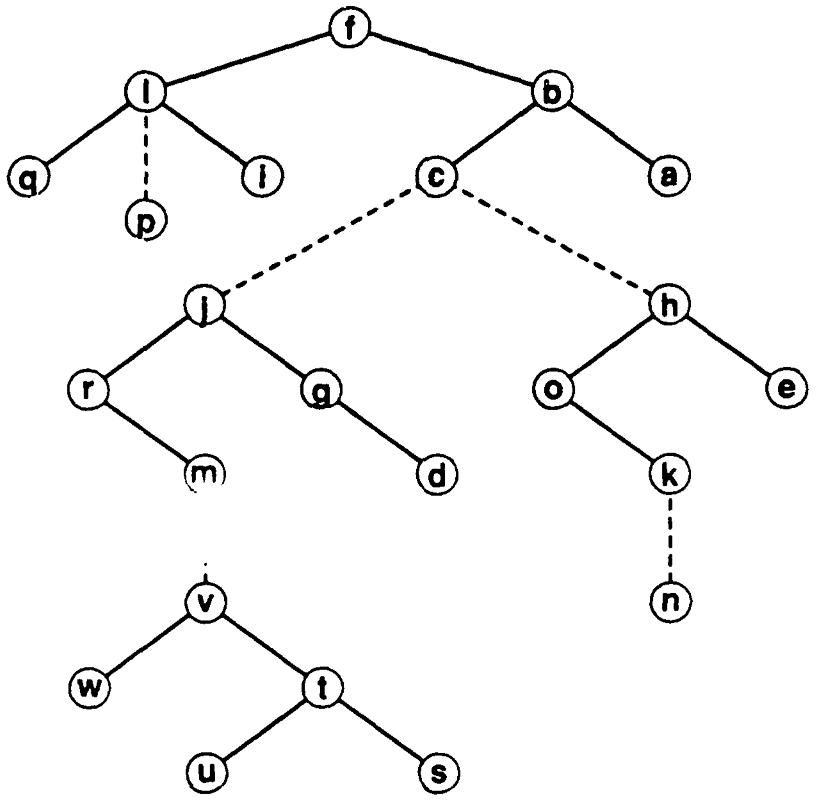


Figure 4b: The virtual tree representing the actual tree of Figure 4a [15].

a procedure called an *extended splay at node v*, abbreviated *e-splay at v*. An extended splay at *v* moves *v* to the root of its virtual tree without changing the structure of the actual tree that the virtual tree represents. The e-splay algorithm is described fully in Sleator and Tarjan [15]. Besides rotation, the extended splay uses a second primitive called *splicing*, which exchanges a middle child with the left child of a solid subtree root (Figure 5b).

Reference [15] gives rules for updating the reversal bit and left, right and parent pointers of the nodes affected by a rotation or splice. Since rotation preserves symmetric order, it does not affect the values of *pred*(*v*) or *succ*(*v*). If the rotation replaces the old subtree root *r* with a new root *v*, however, then the values of *leftmost*(*r*) and *rightmost*(*r*) must be copied to *v*, and *pred*(*leftmost*(*r*)) and *succ*(*rightmost*(*r*)) must be updated to point to *v*.

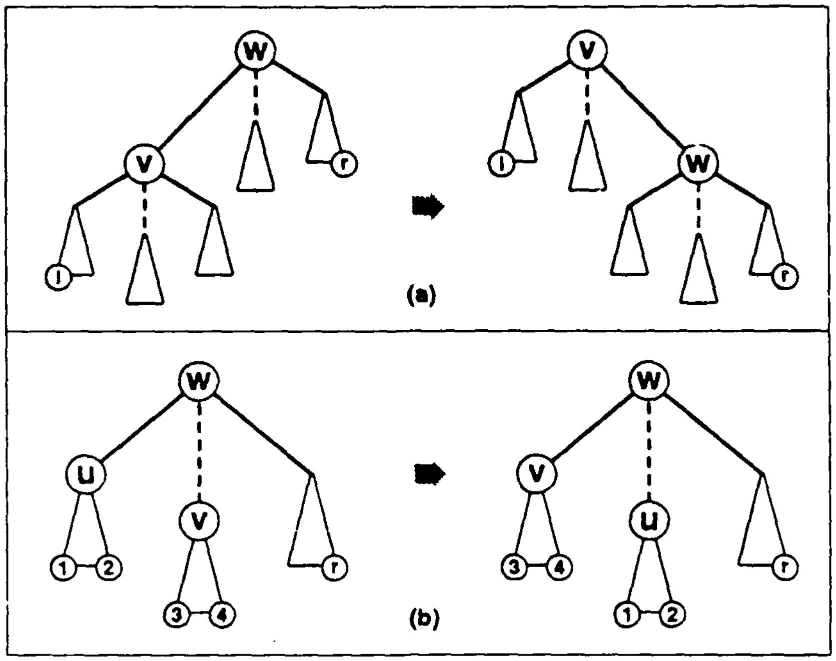


Figure 5: (a) Rotating the edge between v and w. If w is the root of its solid subtree, then pointers to l and r, the leftmost and rightmost nodes in the subtree, must be added to v. (b) Splicing v to w. Node 3 becomes the leftmost node in the solid subtree rooted at w, while nodes 1 and 2 become leftmost and rightmost, respectively, in the solid subtree rooted at u.

If w is the root of a solid subtree, u is the (possibly null) left child of w, and v is a middle child of w, then splicing makes v the left child and w a middle child. The additional pointers introduced in this paper are updated as follows (a prime indicates the new value):

$$\begin{array}{lcllcl}
\mathit{leftmost}'(u) & = & \mathit{leftmost}(w) & \mathit{pred}'(\mathit{leftmost}'(w)) & = & w \\
\mathit{leftmost}'(w) & = & \mathit{leftmost}(v) & \mathit{pred}'(\mathit{leftmost}'(u)) & = & u \\
\mathit{rightmost}'(u) & = & \mathit{pred}(w) & \mathit{succ}'(\mathit{pred}'(w)) & = & w \\
\mathit{pred}'(w) & = & \mathit{rightmost}(v) & \mathit{succ}'(\mathit{rightmost}'(u)) & = & u
\end{array}$$

A *null splice at* w occurs when u is taken to be null. This simply makes the left subtree of w into a middle child without replacing it by another left subtree. (Null splices occur during everts.)

In the above, by $\mathit{pred}(v)$, $\mathit{succ}(v)$, $\mathit{leftmost}(v)$, and $\mathit{rightmost}(v)$ we mean the actual predecessor, successor, etc., of v. The fields containing these values may be switched if node v is in reversed state. During an extended splay at node x, the reversal state is computed for all nodes along the path from x to the root of its virtual tree. The reversal states of the leftmost and rightmost nodes in a solid subtree can be determined by examining which of their predecessor or successor pointers points back to the root. Both rotation and splicing require $O(1)$ pointer steps. The total time for an e-splay is bounded by a constant factor times the number of nodes on the path that is splayed.

In general, the linking together of two virtual trees is implemented by an e-splay in both. When the parent tree is much larger than the child tree, however, we will temporarily defer a full e-splay in the parent tree. To support this deferral process, the nodes of each virtual tree V are partitioned into a collection of disjoint sets, denoted D, which is a coarsening of the partition induced by the solid subtrees. (All nodes in a given solid subtree belong to the same subset, but a set may include many solid subtrees.) These sets are maintained with a fast disjoint set union algorithm. We let $D\text{-}find(u)$ and $D\text{-}union(A,B)$ denote the standard set union operations on this partition. For node u, $N(u)$ must be augmented with an element name for use with $D\text{-}find$.

When *Maketree* creates a new single-node tree, a new set of D is created, containing only the new node. For each set S, the disjoint set union algorithm maintains a *virtual size* denoted $vsize(S)$. A new single set has virtual size 1. Whenever $D\text{-}union(A,B)$ occurs, the algorithm sets $vsize(A) = vsize(A) + vsize(B)$. Since node condensations may reduce the actual number of nodes in a set, we have $vsize(S) \geq |S|$. We denote by $vsize(V)$ the sum of the virtual sizes of the sets into which V is partitioned. Thus $vsize(V) \geq |V|$, with equality when no condensations have occurred. For convenience, we will use "size" to mean virtual size, unless otherwise noted.

If u is a middle child of v and $D\text{-}find(u) \neq D\text{-}find(v)$, the pointer $vparent(u)$ is called a *deferred link*. Deferred links are handled specially during the processes of everting a tree and of linking two trees together. The partition of V imposed by D satisfies the following two *deferred link invariants:*

I. Let v be any node in set S, and let r be the parent node of the first deferred link on the path from v to the root of V. If there is no such deferred link, then let r be the root of V. Then r is the same for all $v \in S$. Thus D defines a unique mapping $D\text{-}root$ such that $D\text{-}root(S) = r$, and the set $S \cup \{D\text{-}root(S)\}$ forms a connected subtree within V. Note that two sets could have the same $D\text{-}root$.

II. If S_2 is the set containing $D\text{-}root(S_1)$ then $vsize(S_2) \geq 2 \cdot vsize(S_1)$ (unless $D\text{-}root(S_1)$ is the root of V, in which case $S_1 = S_2$.

Lemma 3:

Let node v be an ancestor of node u in virtual tree V, and let $n = vsize(D\text{-}find(v))$. Then there are $O(\log n)$ deferred links on the path from u to v in V.

Proof. Straightforward application of invariant II. □

We now give a version of Sleator and Tarjan's extended splay that is modified to handle deferred links. The e-splay is a five pass process. In the first pass, we walk up the path from x to the root of its virtual tree. Each time we encounter a node y that is in a new solid subtree, we splay at y. After the first pass, the path from v to the root consists of dashed edges and deferred links. In the second pass, we walk up the path, splicing at each dashed edge. After the second pass, the path consists of solid subtrees, containing only left children, separated by deferred links. In the third pass, we splay at each parent of a deferred link. After the third pass, the path from x to the root consists only of deferred links. In the fourth pass, we walk up the path, converting each deferred link into a regular dashed edge by uniting the sets that contain the parent and child of the deferred link, and splicing at the resultant dashed edge. After the fourth pass, all nodes on the path belong to the same set of the D partition, and x and the root of the virtual tree are in the same solid subtree. In the fifth pass, we splay at x, making x the virtual tree root. (See Figure 6.)

Lemma 4:

An extended splay preserves the deferred link invariants.

Proof. Only pass four of the e-splay affects the partition defined by D. After pass three, each node on the path belongs to a distinct set, whose D-root is also a path node. Let S' be the union of all sets produced by pass four. Clearly the nodes of S' form a connected subtree, and the root of the virtual tree is $D\text{-}root(S')$. Therefore invariant I still holds.

Let S be any set such that $D\text{-}root(S)$ is contained in S' following pass four. This implies that prior to pass four, $D\text{-}root(S)$ was contained in one of the sets that was united to form S'. Since the virtual size of S' is the sum of its constituent sets, invariant II must hold for set S after pass four. □

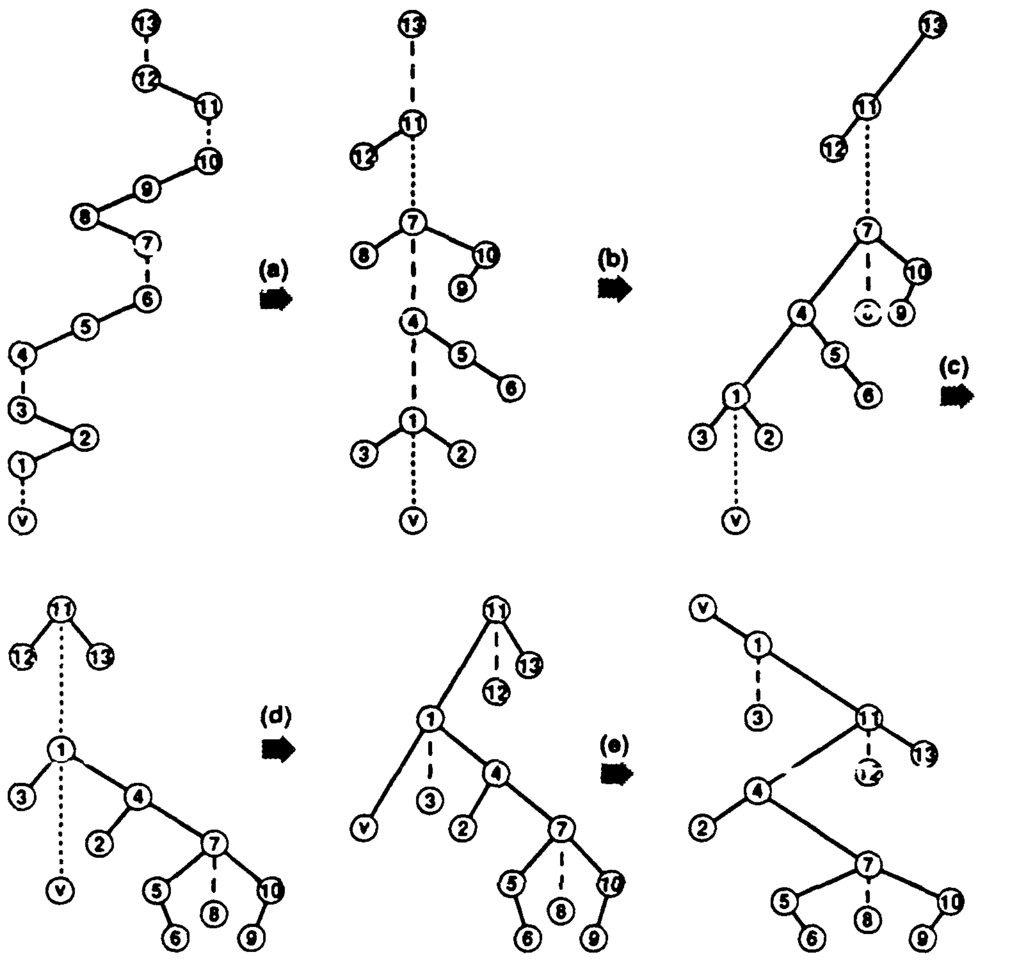


Figure 6: An extended splay at node *v*. Subtrees of nodes on the path have been deleted for clarity. Dotted lines represent deferred links. (a) First pass: splaying inside solid subtrees (see [15] for details of splaying). (b) Second pass: splicing dashed edges. (c) Third pass: splaying solid subtrees between deferred links. (d) Fourth pass: converting deferred links to dashed edges, followed by splicing. (e) Fifth pass: splaying along final solid path.

With the above machinery in place, we turn to the implementation of *Link*(u,v). Basically, we perform an extended splay at u followed by a partial extended splay at v, and then make u a middle child of v. Following the extended splay at u, the right subtree of u contains the path from u to the root of its actual tree. To evert the tree at u, we perform a null splice at u and toggle the reverse bit in u. This reverses the direction of the solid path containing u, making u the root of the actual tree as well as of the virtual tree. (This implementation of evert is described in reference [14]).

Let k be the size of the D set containing u after the extended splay at u. If k is at most half the size of the D set containing v then no e-splay at v occurs and u is simply made a child of v by a deferred link. This preserves the deferred link invariants. Otherwise, a partial extended splay at v is done. To determine the extent of the e-splay, we walk up the tree from v until encountering a deferred link from node x to node y such that the sum of N and the sizes of the sets on the path up to x is at most half the size of the set containing y. We treat x as the root of the virtual tree and do an extended splay at v. This e-splay makes v a child of y by a deferred link, after which u is made a middle child of v and we perform $D\text{-}union(D\text{-}find(u), D\text{-}find(v))$, thereby making the deferred link joining u and v into a dashed edge. If we reach the root of the virtual tree before finding such a deferred link, we simply do a full e-splay, moving v to the root of the virtual tree; then we attach u to v by a dashed edge as described above.

From Lemma 4, the two extended splays preserve the deferred link invariants. It is easily observed that if no e-splay at v occurs, or if the e-splay at v makes v the root of its virtual tree, the link preserves the deferred link invariants. Suppose a partial e-splay at v takes place, terminating at the k^{th} deferred link. Let the k^{th} link pass from node x to node y. Prior to the extended splay at v, node y is the D-root of the set containing x, and after the e-splay y is the D-root of the combined set resulting from the e-splay. After the e-splay, the size of the set containing v is $N + P_{k-1}(v)$, which by the termination condition on the initial walk up from v is at most half the size of the set containing y. Therefore both invariants are preserved in the case of a partial e-splay at v.

Now we turn to the implementation of *Findpath* (u,v). Let T be an actual tree and let V be the virtual tree representing T. *Findpath* (u,v) returns P as a list of pointers to nodes, ordered from u to v, so that two nodes are adjacent in P if and only if they are adjacent in T. As in section 2, we find a path from u to v by walking up the paths from u and from v to the root of the real tree T. We generate the two paths in lock-step and stop as soon as they intersect at the nearest common ancestor of u and v. The general strategy to find the path from node u to the root of T is to locate u in V, and follow successor pointers until reaching the rightmost node r in the solid subtree containing u. Then we jump to the subtree root t via *succ* (r) and follow the dashed edge *vparent* (t) to the next node on the path. This next node lies in a new solid subtree. We repeat this procedure until encountering a null *vparent* (t) pointer. Then r is the real tree root.

Unf‹ .:unately, this strategy cannot be applied directly, since it is impossible to determine the reversal state of u, and hence the interpretation of *pred*(u) and *succ*(u), without walking down to u from the solid subtree root. This would increase the cost of path finding to $O(d+|P|)$, where d is the depth of u. It is possible, however, to generate a path that consistently moves in the same direction once a direction in which to start is chosen. If we follow a pointer from node x to node y, then one of *pred*(y) or *succ*(y) must point back to x. To generate the next step in the path, we can simply follow the other pointer. Furthermore, once the path encounters the leftmost or rightmost node in the solid subtree, we can jump to the subtree root and determine whether the path goes up or down the actual tree.

Given this, we can find the path P between nodes u and v using the following algorithm. We march in lock-step in both directions out of u and v, marking nodes encountered. Thus in parallel we generate two pairs of tentative paths, one node at a time. When one tentative path encounters a solid subtree root, we determine which of the two tentative paths goes up the tree, and discard the other tentative path, after unmarking the incorrectly marked nodes. If the subtree root was encountered in the incorrect direction, then we then proceed in the correct direction (in step with the other search) until finding the subtree root again. Then we follow the parent pointer of the root into a new solid subtree, at which point ambiguity again arises. Path generation is complete when one tentative path from u intersects one tentative path from v. By marching in two directions simultaneously, we double the number of pointer steps required to generate one node of the path, but the total time to generate a path of length l remains $O(l)$.

Knowing a way to find P, we can turn to the implementation of *Condensepath*(P,A). P consists entirely of round nodes, all of which must be condensed into a single round node. Let *Condense2path*(x,y,A) be a function that condenses a path consisting of two adjacent nodes x and y, and labels the resultant node z by A. *Condensepath*(P,A) proceeds by repeating the following *condensation step* until P consists of a single node: select a pair of adjacent nodes x,y from P. Perform *Condense2path*(x,y,A). (Let T' be the tree resulting from this condensation.) Replace x and y in P with z.

Since path condensation preserves adjacency, it is clear that after a condensation step, P is the path between u and v in tree T'. An inductive proof using this observation shows that *Condensepath* correctly condenses a path of arbitrary length. We now turn to the implementation of *Condense2path* (x,y,A). We distinguish two cases based on the relationship between x and y in the virtual tree: 1) x and y are in the same solid subtree; and 2) x and y are in different solid subtrees. In Case 1, at least one of *pred* (x), *succ* (x) contains a pointer to y. In case 2, neither of these fields contain a pointer to y. This fact can be used to determine which case applies.

Case 1 is shown in Figure 7a. The adjacency of x and y in the actual tree implies that one is the ancestor of the other in the solid subtree. Assume that y is the ancestor of x. For the moment, we assume that the reversal states of x and y are known. Let y be the successor of x. (The other possibility is symmetric.) This implies that the right subtree of x is empty, and that the successor pointer of x points to y. By examing the fields of x and y for this configuration, we can determine which of them is in fact the ancestor. That is, y is the ancestor of x if *right* (x) is null and *succ* (x) is y, or, in the symmetric case, *left* (x) is null and *pred* (x) is y.)

Let $w = vparent(x)$. (It may be the case that $w = y$.) Let $u = left(x)$. To begin the condensation, node u is made a child of w in place of node x. This is done by replacing the pointer to x in w by a pointer to u and setting $vparent(u) = w$. The old value of *reverse* (u) is replaced by the exclusive-or of *reverse* (u) and *reverse* (x). Nodes x and y are combined to form z. The fields in $N(z)$ are copied from $N(y)$, with the exception of *label* (z), which is set to A, and of *pred* (z), which is copied from $N(x)$.

Combining x and y, which unites their subnode sets, guarantees that all former pointers to x or y are now pointers to z. The former predecessor of x and the former successor of y become the predecessor and successor, respectively, of z. All former middle children of x or y become middle children of z. Hence, node adjacencies in the transformed tree are correctly preserved. In general, the reversal states of x and y are not known, but they are not actually needed. One of *left* (x) or *right* (x) must be null, so the pointer to u can be found in the other. Similarly, one of *pred* (y) or *succ* (y) points to x. The corresponding field in z is copied from whichever of *pred* (x) or *succ* (x) does not contain a pointer to y.

Case 2 is shown in Figure 7b. The adjacency of nodes x and y implies that one, say x, must be rightmost in its solid subtree. The root of this solid subtree is a middle child of y. The reversal state of x can be computed by determining which of *pred* (x) or *succ* (x)

points to the root of the solid subtree. Since x is rightmost, its right subtree is empty. Let $u = left(x)$. If x is the root of its subtree, a null splice is performed to make u a middle child of x. Otherwise, let $w = vparent(x)$. Using the pointer and reversal bit updates given in case 1, u replaces x as the right child of w. This makes $pred(x)$ rightmost in the solid subtree. The leftmost field in the solid subtree root and the successor field in $pred(x)$, (i.e., the field containing a pointer to x) must be updated accordingly. Finally, x and y are condensed to give z. $N(y)$ is copied to $N(z)$, and z is labeled A.

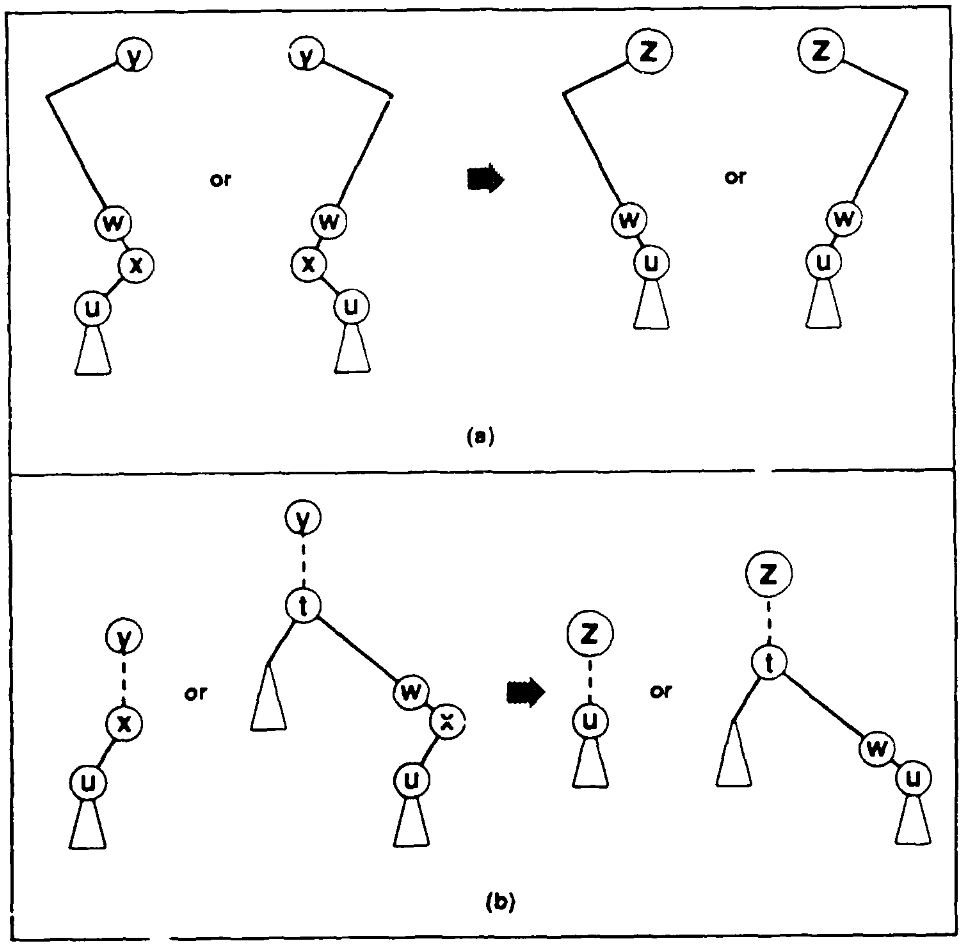


Figure 7: a) Case 1 of *Condense2path*(x,y,A). $w = vparent(x)$. (We may have $w = y$.) Node z is the result of the condensation. b) Case 2 of *Condense2path*(x,y,A). (We may have $t = w$.)

Lemma 5:

Condense2path(x,y,A) preserves the deferred link invariants.

Proof. If x and y initially belong to the same set S of D, then the condensation does not change the invariants for S. If, prior to the condensation, x (or, equivalently, y) is D-$root(S')$ for some set S', then all nodes of S' are descendants of one or more children of x. The condensation makes all children of x (and y) into children of z. Hence invariant I is preserved for S', with D-$root(S') = z$. Although the condensation decreases the actual size of S by one, its virtual size is unchanged. Therefore invariant II is preserved for all sets whose D-root is contained in S.

In case 1 of *Condense 2path*(x,y,A), x and y belong to the same solid subtree, and so must belong to the same set S. In case 2, however, x and y may belong to two different sets, say S_1 and S_2, respectively. This situation occurs when the root of the subtree containing x is a child of y by a deferred link. The invariants imply that y is D-$root(S_1)$ and $vsize(S_2) \geq 2vsize(S_1)$. For any descendant of x that belongs to S_1, the deferred link between the solid subtree root and y is the first deferred link on the path to the root of the virtual tree. The condensation makes all middle children of x into middle children of z by deferred links, while the left subtree of x remains in the solid subtree that contained x. Thus after the condensation z is the parent of the first deferred link on the path from any element in S_1 to the virtual tree root. Since no virtual size changes, invariants I and II are preserved for S_1 and S_2, with D-$root(S_1) = z$. As before, invariants I and II are preserved for all sets whose D-root is contained in S_1 or S_2 prior to condensation. □

The following lemma follows from an examination of the *Condense 2path* algorithm.

Lemma 6:

During condensation, the number of virtual tree descendants of any node is non-increasing.

5. Amortized Analysis of Link/Condense Operations

To begin the running time analysis, we observe that the time to perform *Link*(u,v) is a function of the original depth of u and the length of the path traveled up from v. We can measure this depth by the number of rotations performed during the two e-splays. The time to perform *Findpath*(u,v) is determined by the number of times a node along the path is found, and the time to perform *Condensepath*(P,A) is measured by the number of times a pair of nodes are combined by *Condense 2path*. We define rotation, node-finding, and pairwise combination to be basic primitives of cost 1, since each is responsible for at most a constant number of condensible node operations. The cost of a link/condense tree operation is measured by the number of primitives executed while performing the operation, and the amortized time for an operation is the product of the cost of the operation with the amortized time to perform $O(1)$ condensible node manipulations.

The amortized analysis uses a *potential function* [15,17] defined on the virtual tree structure. For each node x in a virtual tree, we define the *weight of node* x, $w(x)$, as the number of virtual tree descendants y of x, including x itself, such that $D\text{-}find(y) = D\text{-}find(x)$, i.e. all descendants of x belonging to the same D-set as x. We define the *logarithmic weight* $\mathrm{lgw}(x)$ of node x to be $\log w(x)$.† The potential Φ is defined as follows:

$$\Phi = \sum_{nodes\ x} 2\mathrm{lgw}(x) + \sum_{S \in D} 18(3 + \log vsize(S))$$

Let Φ_i denote the value of the potential function after the ith operation. If t_i is the actual cost of the ith operation then the amortized cost a_i of the ith operation is defined by

$$a_i = t_i + \Delta\Phi \text{ , where } \Delta\Phi = \Phi_i - \Phi_{i-1}$$

We begin by analyzing the cost of an extended splay at node v. Suppose that there are k deferred links on the path from v to the root of the virtual tree (or, in a case of a partial e-splay, to the node that is treated as the root). Then the path passes through $k+1$ sets $S_0, S_1, \cdots, S_k$ of the D-partition, where S_0 contains v and S_k contains the virtual tree root. Abbreviating $vsize(S_i)$ by N_i, let $p_i = \sum_{0 \le j \le i} |S_j|$, and $P_i = \sum_{0 \le j \le i} N_j$. Since $N_i \ge |S_i|$ for all i, we have $P_i \ge p_i$.

Lemma 7:

$P_i \le 2N_i$ for $0 \le i \le k$.

Proof. From Lemma 3, if $N_i = m$, then $i \le \log m$. By invariant II, $N_{i-1} \le N_i/2$, $N_{i-2} \le N_{i-1}/2$, etcetera. Thus

$$\sum_{0 \le j \le i} N_j \le \sum_{0 \le j \le \log N_i} \frac{N_i}{2^j} \le 2N_i - 1.$$

† By log we mean continuous binary logarithm.

□

Lemma 8:

The amortized cost of an extended splay at v is at most $24\log N_k + O(1) + \sum_{0 \le i \le k} [(12\log N_i + 4) - 18(3 + \log N_i)]$, where k and N_i are defined as above.

Proof. Passes one, two, three, and five do not affect the partition defined by D, so any changes to the potential in these passes occur only in the term involving lgw(x). In reference [15], Sleator and Tarjan use this reduced potential function in analyzing their splay and extended splay algorithms. From their paper we draw two facts: first, the cost of the splay in pass five is at most $6\log p_k + 1$; second, the cost of passes one through three in the contiguous section of path between the i^{th} and $i+1^{st}$ deferred links is at most 12lgw(x_i)+2, where x_i is the i^{th} node on the path after pass three. Thus the cost of passes one, two, three, and five is

$$6\log p_k + 1 + \sum_{0 \le i \le k} [\, 12\,\mathrm{lgw}(x_i) + 2 \,].$$

After pass three, the path from v to the root consists of the $k+1$ nodes x_i. Pass four does no rotations so it has zero actual cost, but the unions done in pass four change the potential. The $k+1$ sets S_i, $0 \le i \le k$, are replaced by a single set S' with virtual size P_k. The unions also increase the weight of each node x_i on the path, since the number of descendants of x_i that belong to the same set as x_i increases. By invariant I, node x_i is D-$root(S_{i-1})$ for $1 \le i \le k$. This implies that only the nodes on the path increase in weight as a result of the unions. During pass four node x_i increases in weight by at most $p_i - w(x_i)$, where (x_i) is the weight of x_i prior to pass four. Thus the cost of pass four is at most

$$18(\log P_k + 3) + \sum_{0 \le i \le k} [\, 2\log(p_i) - 2\,\mathrm{lgw}(x_i) - 18(3 + \log N_i) \,]$$

We have $P_i \ge p_i \ge w(x_i)$ and $N_i \ge w(x_i)$. From Lemma 7, $2N_i \ge P_i$. Combining terms, and using these observations, the lemma follows. □

Lemma 9:

Let u be contained in virtual tree V. The amortized cost of $Link(u,v)$ is $O(\log n)$, where $n = vsize(V)$.

Proof. Using Lemma 8, it is straightforward to show that the extended splay at u has cost $O(\log n)$, since each term of the sum in Lemma 8 is less than zero, and $N_k \le n$. If no e-splay at v occurs, this is the entire cost of the link. To analyze the cost of an e-splay at v, we divide the sum of Lemma 8 into two parts. Let l be minimal such that $N_l \ge n$. We rewrite the sum of Lemma 8 as $A + B$, where

$$A = 12\log N_l + 4 + \sum_{0 \le i < l} [(12\log N_i + 4) - 18(3 + \log N_i)]$$

and

$$B = 6\log N_k + O(1) + \sum_{l < i \le k} [(12\log N_i + 4) - 18(3 + \log N_{i-1})].$$

To bound these sums, we observe that since the e-splay at v did not terminate upon reaching the i^{th} set, it must be the case that $n + P_{i-1} > N_i/2$. Each term of the sum in A is at most zero, so A is $O(\log N_l)$. We bound N_l as follows: by Lemma 7, $P_{l-1} \le 2N_{l-1}$. By the definition of l, $n > N_{l-1}$. Together with our initial observation, this implies that $N_l < 6n$, and hence $A = O(\log n)$.

To bound B, we observe that by the definition of l, $N_i \ge 2n$ for $i > l$. Together with Lemma 7, this implies that $5N_{i-1} \ge 2(n + P_{i-1}) > N_i$. Using this inequality, we find that each term of the sum in B is at most $-(6\log N_i - 8)$, and hence B is $O(1)$. Therefore, the extended splay at v has amortized cost at most $O(\log n)$. Finally, making u a middle child of v, and uniting the two sets containing u and v, increases the potential of v by at most $O(\log n)$. Thus *Link* (u,v) has amortized cost $O(\log n)$. □

Theorem 3:

A sequence of m bridge-connected component operations can be performed in worst-case time $O(m\alpha(m,n))$.

Proof. As before, each condensible node operation takes amortized time $O(\alpha(m,n))$. A call to *Maketree* costs $O(1)$, since it increases the potential by a constant amount. Path finding does not modify the tree, so the potential is unchanged. Since a *find block* operation requires the finding of a constant-length path, the total time spent in *find block* operations is $O(m\alpha(m,n))$. Lemmas 5 and 6 imply that *Condense2path* does not increase the potential and so has amortized cost at most 1. Thus the total cost of condensing edge insertions is $O(n)$, and the worst-case time is $O(n\alpha(m,n))$.

To determine the total cost of component links, we note that since the number of square nodes bounds the number of round nodes, the virtual size of a bridge-block tree is at most twice the number of vertices in the corresponding component of the graph. Combining this observation with Lemmas 7 and 8, we find that the amortized cost of a component link is $O(\log n)$, where n is the number of vertices in the smaller component. This gives the following recurrence for the total cost of component links:

$$T(n) \leq \max_{1 \leq j \leq n/2} \{T(j) + T(n-j) + c(1+\log j)\}$$

This implies $T(n) = O(n)$. There is at most one call to *D-find* or *D-union* per pointer step, and hence the total time spent manipulating the D partition is $O(n\alpha(T(n),n))$, which is $O(m\alpha(m,n))$. Thus the total cost of processing component links is $O(m\alpha(m,n))$. □

In conclusion, we remark that all data structures are size $O(n)$, and thus the space required by the algorithm is $O(n)$.

6. A Data Structure for an Optimal Block Algorithm

The link/condense tree data structure of the previous section can be modified to produce an efficient data structure for maintaining the block structure forest. Unfortunately, these modifications are not trivial. The complications arise from the fact that a block tree path consists of alternating round and square nodes, and during path condensation only the round nodes are combined. Care must be taken in restructuring the virtual tree so that movement of the square nodes does not increase the value of the potential function (which would cause the amortized time per update to increase).

We take as our starting point the complete link/condense tree data structure of the previous section. On this data structure, we impose the following *solid subtree restrictions:*

(i). *Each solid path in the actual tree either is a single round node or terminates in a square node (i.e., a square node is rightmost in the corresponding solid subtree of the virtual tree.)*

(ii): *All square nodes are leaf nodes of their solid subtrees.*

The algorithm for *Findpath*(u,v) given in Section 4 can be used here without changes. To implement *Condensepath*(P,A), we use the function *Condense3path*(r,u,s,A), which transforms a three-node path consisting of round node r, square node u, and round node s, and returns the round node t, labeled A, resulting from the condensation of r and s. Path condensation proceeds by repeatedly selecting a three-node path from P and replacing it with the single node returned by *Condense3path.* The process terminates when P consists of a single round node. There are four cases for *Condense3path*(r,u,s,A):

1) all three nodes are in the same solid subtree;

2) u and one round node are in one subtree whose root is a middle child of the other round node;

3) u and one round node are in one subtree, and the other round node is a middle child of u;

4) all three nodes are in different solid subtrees.

Which case applies can be determined by examining *vparent* pointers.

In cases 1 and 2, the relationship between virtual tree and actual tree implies that in the actual block tree, one round node is an ancestor of u and the other is a descendant of u. In case 1, restriction (i) implies that in the solid subtree one of r and s is a descendant of the other, and u is a child of the descendant round node. This fact can be used to determine which of r and s is the ancestor. Let r be the descendant. (The subcase with r the ancestor is analogous.) Node u is either the left or right child of r. The other child of r is made a child of *vparent*(r) using the pointer and reverse bit updates described in case 1 of the *Condense2path* routine of Section 4. Then nodes r and s are condensed together to give node t, with the fields of t being copied from the fields of s. This makes u a middle child of the condensed node.

In case 2, assume that r is in the same solid subtree as u. (The case of s in the same solid subtree is analogous.) Restriction (i) implies that u is the right child of r and that u is rightmost in its solid subtree. As in case 2 of *Condense2path*, the left child of r is made into the right child of *vparent*(r). Then nodes r and s are condensed together to give node t, with the fields of t being copied from the fields of s. This makes u a middle child of the condensed node.

Cases 3 and 4 are simpler. At least one of the round nodes r and s is a middle child of u. By restriction (ii) this round node must be a singleton solid subtree. The two round nodes are simply condensed to give t. The fields of t are copied from whichever round node is not a descendant of u, or, if both r and s are descendants, from either one.

The four cases are shown in Figures 8a through 8d. From examination of the cases, it is clear that *Condense 3path* does not violate the solid subtree restrictions. The proofs of Lemmas 5 and 6 in Section 4 can be adapted to show that *Condense 3path* preserves the deferred link invariants, and does not increase the weight of any node.

We now turn to the implementation of *Link*(u,v). Although *Link* remains unchanged in basic design, the existence of solid subtree restrictions necessitates changes to the extended splay procedure, and this in turn causes slight modifications to *Link*.

Recall that the extended splay is based on *splaying*. A splay at node x moves x to the root of its solid subtree by rotating every edge along the path from x to the root. Splaying does not rearrange the subtrees rooted at nodes off the path from x to the root. This implies that if x is round, a splay at x cannot violate restrictions (i) or (ii). Splaying a square node to the root, however, will violate restriction (ii) except in the trivial case that the square node is a singleton solid subtree. Therefore, we define the function *square-splay*(v), which moves a square node to within one step of the root.

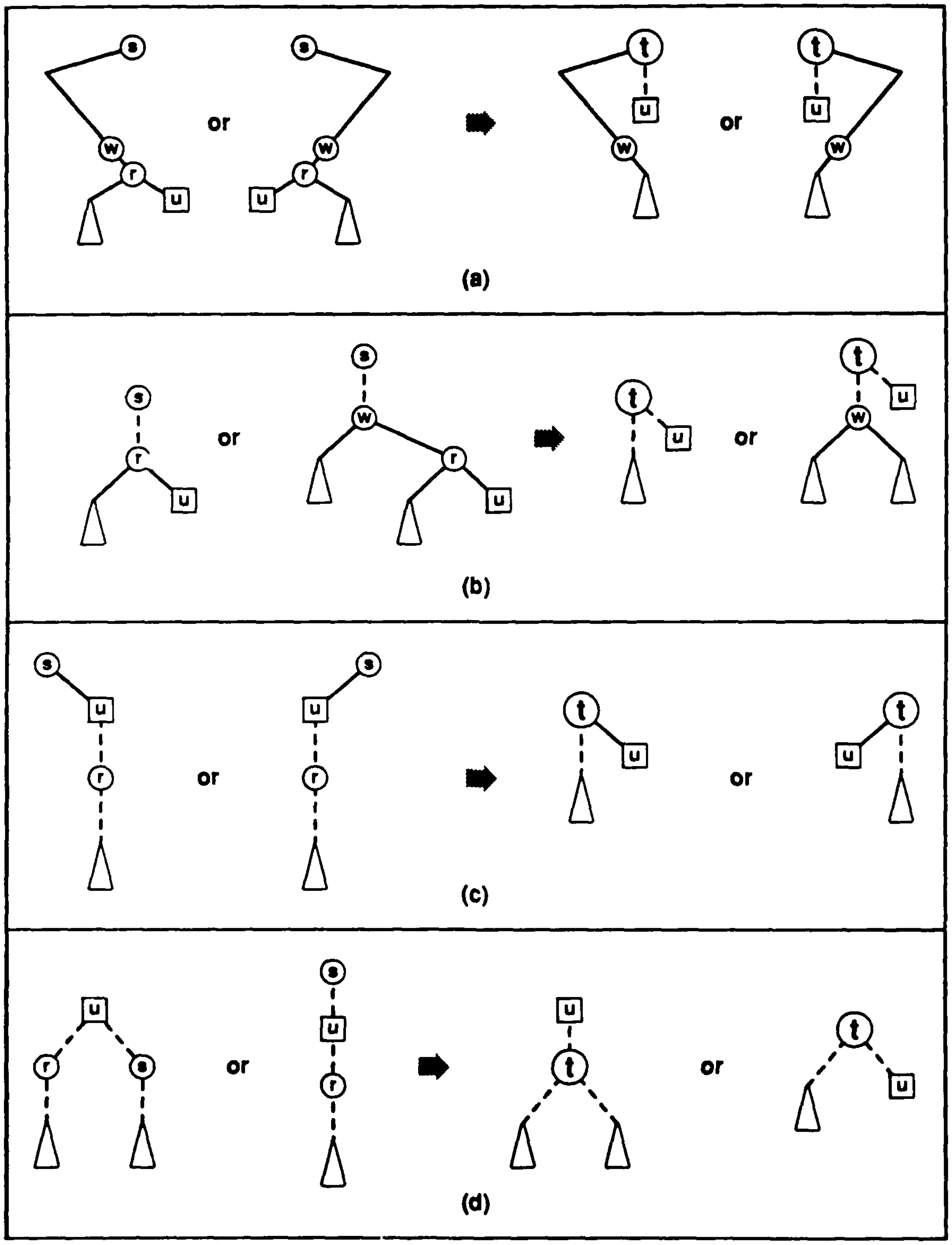


Figure 8: Cases 1 through 4 of *Condense* 3*path* (r,u,s,A).
Node t is the result of condensing nodes r and s.

```
square-splay ($v$) begin
    splay at pred ($v$); splay at succ ($v$);
    rotate the edge between $v$ and pred ($v$);
    perform a null splice at pred ($v$);
    perform a null splice at $v$;
end
```

In the above, by *pred*(v) and *succ*(v) we mean the actual predecessor and successor, respectively, of v. The fields containing these values may be switched if node v is reversed. (The reversal states of all nodes along the splay path are computed by an initial walk down the path.) The behavior of splaying is such that after the first two splays, v is a right child of *pred*(v) and *pred*(v) is a left child of *succ*(v) (see [15]). If v has no predecessor or successor, then the code involving the missing node can simply be ignored. For example, if v has no predecessor, then we need only splay at the successor. At the conclusion of *square-splay*(v), node v remains a leaf; either it is a singleton solid subtree or it is the left child of the root. Thus *square-splay* does not violate the solid subtree restrictions.

We must also be careful when splicing. An attempt to splice a round child to a square parent may violate restriction (ii), while an attempt to splice any child into a round node that forms a single-node subtree may violate restriction (i). To handle the first case, we introduce a new splice function, *square-splice* (r), where r is a round child of a square parent. Note that by restriction (i), r is a single-node solid subtree.

```
square-splice ($r$) begin
    let $v$ = vparent ($r$); splice $r$ to $v$.
    rotate the edge between $r$ and $v$;
end
```

At the conclusion of *square-splice* (r), square node v is the right child of r. Since r had no right child initially, v remains a leaf node after the rotation. Thus *square-splice* does not violate the solid subtree restrictions.

In the second case, we can allow the splicing of a left subtree to a round node r that forms a single-node subtree as long as r has a parent. Since this parent must be a square node, r will immediately participate in a square-splice that will give it a square right child, guaranteeing that restriction (i) is not violated. The splice is disallowed if r is the root of the virtual tree (and hence the root of the actual tree), or if there is a deferred link from *r* to its parent.

As before, an extended splay at *v* is a five-pass process. (In the block algorithm we only e-splay at square nodes.) We use square-splays and square-splices as needed, and do not splice at singleton round nodes from which there are deferred links. After the first pass, the path from *v* to the root consists of dashed edges, deferred links, and solid edges between a square node and its parent. After pass three, the path from *v* to the root consists primarily of deferred links. Between each pair of deferred links there is a section of path that consists of either a single round node, or a solid edge between a square node and its round parent, or a solid edge between a square node and its round parent followed by a dashed edge leading into a single-round-node round subtree. After passes four and five *v* is either at the root, is a left or middle child of the root, or is the left child of a middle child of the root. Figure 9 gives an example of the first three passes of an extended splay, showing square-splays and square-splices.

The *Link* algorithm is essentially the same as that of Section 4. The only difference is in the way a tree is everted at square node *v*. Consider the possible cases after the initial e-splay at *v*. If *v* is the root of its virtual tree, no processing is needed to do the eversion. If *v* is the left child of the virtual tree root, then we simply toggle the reverse bit in the root. If *v* is a middle child of the root, we perform an ordinary splice at *v* and toggle the reverse bit in the root. If *v* is a left child, and *vparent*(*v*) is a middle child of the root, then we perform an ordinary splice to make *vparent*(*v*) a left child of the root, and toggle the reversal bit in the root. In the latter three cases, *v* becomes the end of a solid subpath, so restriction (i) is not violated.

The modifications to splaying and splicing do not affect the way the deferred link partition of the virtual tree is maintained. We may therefore use the proof of Lemma 4 in Section 4 to conclude that e-splay and *Link* preserve the deferred link invariants. With our description of the block algorithms complete, we turn to the analysis of their running times.

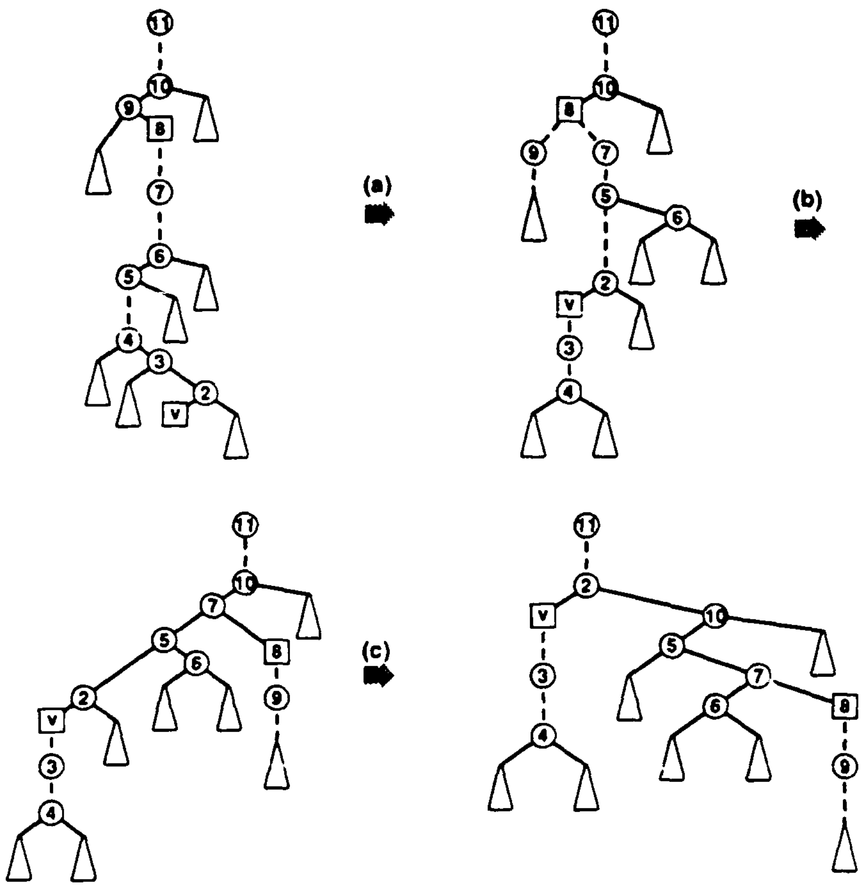


Figure 9: Passes 1-3 of an extended splay at node *v*. The path contains no deferred links. Triangles represent subtrees with a square rightmost node. (a) First pass: splaying and square-splaying inside solid subtrees. (b) Second pass: splicing and square-splicing dashed edges. (c) Third pass: square-splay along final solid path.

7. Amortized Analysis of the Block Link/Condense Tree

In general our analysis here closely follows that of Section 5. We define the potential Φ to be:

$$\sum_{\text{nodes } x} 5\lg w(x) + \sum_{S \in D} 85\,(3 + \log vsize(S))$$

It is straightforward to adapt the analysis of Section 5 to show that the total time spent in queries and condensing edge additions is $O(m\alpha(m,n))$. To achieve the same bound for component links, it suffices to demonstrate that the amortized cost of *Link*(u,v) is still $O(\log n)$. As before we measure cost by number of rotations. We will be fairly terse in the analysis, relying on results proved in reference [15]. The reader is advised to examine that paper for further details and explanation.

We begin by analyzing the costs of *square-splay*(v) and *square-splice*(r). From Lemma 1 of reference [15] we draw the following fact: let x be a left or right child of y. A single rotation of the edge between x and y has amortized cost $1 + 3(5\lg w(y) - 5\lg w(x))$. This bounds the cost of a square-splice.

From the same source, we draw a second fact: let y be the root of a solid subtree that contains node x. The amortized cost of a splay at x is at most $1 + 3(5\lg w(y) - 5\lg w(x))$. In executing *square-splice*(v) we perform two splays and a single rotation. The properties of symmetric order imply that, since v is a leaf, both the predecessor and the successor of v must at all times be ancestors of v, and hence their logarithmic sizes are always greater than that of v. This implies that the amortized cost of *square-splay*(v) in a solid subtree rooted at t is at most $3 + 45(\lg w(t) - \lg w(v))$.

We now bound the cost of the first three passes of an extended splay at node v.

Lemma 10:

Consider a section of the path from v to the root that is bounded by deferred links, i.e. all nodes on the path are in the same set of the D-partition. Let x be the lowest node on the section of path and let t be the highest node. (Thus x is a parent of a deferred link and t is a middle child of a deferred link.) The cost of passes one, two, and three in this section of path is is at most $75\log n + 13$, where $n = w(t)$.

Proof. Let k be the number of solid subtrees on the initial path from x to t. The amortized cost of the first pass is at most $3k + 45\lg w(t)$. This bound follows from summing the splay or square-splay costs within each solid subtree. The amortized cost of the second pass is at most $k + 15\lg w(t)$, since there are most k square-splices and the sum again telescopes. Thus the amortized cost of first two passes is $4k + 60\lg w(t)$.

If x is square, then after the second pass, x is at depth k, and in the third pass, *square-splay*(x) will perform exactly one splay, moving the parent (successor) of x to the root (or to within one step from the root, if t forms a round single-node subtree). If x is round, x is at depth $k-1$ and is itself splayed to the root in the third pass. Therefore, at least $k-2$ rotations occur in the third pass. We charge 5 for each of these $k-2$ rotations;

the additional charge accounts for $4k-8$ of the rotations left over from the first two passes. From the discussion of e-splay in reference [15], we find that even with this additional charge, the amortized cost of the final splay is at most $5 + 15\,\lg w(t)$. Summing over the three passes, we obtain the desired result. The constant factor can be reduced to 45 by noticing that single-round-node subtrees and square nodes come in pairs along the path, and j such pairs cause at most $2j$ extra rotations in passes one and two. □

The D-unions that occur in pass four are more expensive here than in the bridge-block algorithm, because there may be as many as three nodes between each deferred link, each of which has its weight increased by the D-unions. Using the definitions of Lemma 8, Section 5, and letting x_i denote the highest node on the path belonging to set S_i, we find that the D-unions change the potential by at most

$$85(\log P_k + 3) + \sum_{0 \le i \le k} [\, 15\log(p_i) - 5\lg w(x_i) - 85(3 + \log N_i)\,]$$

After the D-unions in pass four, the path from v to the root contains no deferred links, so we can use Lemma 10 to bound the cost of the remaining splices in pass four and the splay in pass five. Therefore, a bound on the amortized cost of an extended splay is given by

$$160\log N_k + O(1) + \sum_{0 \le i \le k} [(85\log N_i + 28) - 85(3 + \log N_i)]$$

As in the proof of Lemma 9, Section 5, this bound can be used to show that $Link(u,v)$ has amortized cost $O(\log n)$, where n is the virtual size of the tree containing u.

Theorem 4:

Any sequence of m biconnected component operations can be performed in worst-case time $O(m\alpha(m,n))$.

8. General Lower Bounds

Lower bounds for the on-line block and bridge-block problems can obtained using simple reductions from the disjoint set union problem. Let n be the number of elements and m the number of operations in an instance of disjoint set union. Tarjan [16] gave a lower bound of $\Omega(\alpha(m,n))$ on the amortized time per operation and Blum [2] gave a lower bound of $\Omega(\log n/\log\log n)$ on the worst-case time of a single operation. Both these lower bounds apply to the class of *separable pointer algorithms* for set union. This class is described more fully in references [2,8,13,16,23]; in summary, a separable pointer algorithm uses a linked data structure that can be represented by a directed graph. The separability rule states: "after any operation, the data structure can be partitioned into node-disjoint subgraphs, one corresponding to each currently existing set and containing all the elements in the set. The name of the set occurs occurs in exactly one node in the subgraph. *No edge leads from one subgraph to another.*"[23]

Recently, Fredman and Saks [6] have given an $\Omega(\alpha(m,n))$ bound on the amortized cost per operation in the cell probe model of Yao [24]. It is also possible to show an $\Omega(\log n / \log\log n)$ bound on the worst-case cost per operation [M. Fredman, private communication, 1989]. In this powerful and general model, memory is organized into cells, each of which can hold $\log n$ bits. In answering a query, a cell-probe algorithm is allowed to randomly access cells based on the information gathered from previous probes. The two models are related as follows: if the number of bits in a separable pointer machine node is bounded by $\beta(n) \geq \log n$, then the separable pointer machine can be simulated by a cell probe machine, with running time increasing by a factor of $\beta(n)/\log n$.

The reduction from disjoint set union to the on-line block and bridge-block problems is straightforward. For each set element a we create a pair of vertices a_0 and a_1 connected by an edge e_a. To answer a find query, we execute *find block*(a_0) or *find block*(a_0,a_1), depending on whether the reduction is to the bridge-block or block problem, respectively. To unite the sets containing elements a and b, we add an edge between a_0 and b_0 and an edge between a_1 and b_1. Thus each set corresponds to a component that is both bridge-connected and biconnected. We define a separable pointer algorithm for bridge-connectivity or biconnectivity to be an algorithm that uses a linked data structure in which no pointer connects the subgraphs representing each graph component. If we begin with a separable algorithm for bridge-connectivity or biconnectivity, the above reduction gives a separable pointer algorithm for disjoint set union. Similarly, the reduction can be used to give a cell probe algorithm for disjoint set union.

We can also give a reduction of disjoint set union to the variant of the block problem in which the graph is initially connected. The initial graph resembles a wheel with hub vertex h. For each element a there is a vertex v_a and an edge connecting v_a to h. Queries are answered by executing *find block*(h,v_a). To unite the sets containing a and

b, an edge is added between v_a and v_b. The algorithm given by this reduction does not fit the separable pointer machine model, but the cell probe lower bounds apply. For the variant of the bridge-block problem in which G is initially connected, we know only the trivial lower bound of $\Omega(1)$ on the time per operation.

9. Remarks.

One major difference between the dynamic tree data structure of Sleator and Tarjan and the link/condense tree data structure presented here is that in the latter the time to link together two trees is dependent only on the size of the child tree. If condensation is not required, the tree can be modified to implement all the standard dynamic tree operations, such as *find min* and *add cost*, in time $O(\log n)$, while still allowing fast linking. Such a tree would be suitable for any tree-based algorithm in which a recurrence relation similar to that of the bridge-block or block algorithm arises.

This paper leaves open the problem of maintaining the triconnected components of a graph undergoing edge insertion, and it leaves a gap in the table of section 1 between the upper and lower bounds for the bridge-block problem on an initially connected graph. We also leave open the problem of implementing any kind of edge deletion in time $o(n)$ per operation. Reif [12] introduces the notion of *complete dynamic problems*, i.e., a collection of problems with the property that if one problem can be solved in $o(n)$ time per operation, then all the problems can be solved in $o(n)$ time per operation. Examples of these problems include acceptance by a linear-time Turing machine, the Boolean circuit evaluation problem, and the depth-first search numbering of a graph. These problems seem to have no better on-line solution than simply running a linear-time off-line algorithm whenever the input instance changes. One can ask whether bridge-connectivity and biconnectivity are complete dynamic problems.

References

[1] B. Awerbuch and Y. Shiloach, "New connectivity and MSF algorithms for shuffle-exchange network and M," *IEEE Trans. on Computers* C-36 (1987), pp. 1258-1263.

[2] N. Blum, "On the single-operation worst-case time complexity of the disjoint set union problem," *SIAM J. Comput.* 15 (1986), pp. 1021-1024.

[3] G. A. Cheston, "Incremental algorithms in graph theory," Tech. Rep. No. 91 (PhD. Diss.), Dept. of Computer Science, University of Toronto, (1976).

[4] G. Di Battista and R. Tamassia, "On-Line Planarity Testing," Tech. Rep. No. CS-89-31, Dept. of Computer Science, Brown University (1989).

[5] S. Even and Y. Shiloach, "On-line edge deletion," *J. Assoc. Comp. Mach.* 28 (1981), pp. 1-4.

[6] M. L. Fredman and M. E. Saks, "The Cell Probe Complexity of Dynamic Data Structures," *Proc [illegible] ACM Symposium on Theory of Computing*, Seattle, Washington (May 1989), pp. 345-354.

[7] G. N. Frederickson, "On-line updating of minimum spanning trees," *SIAM J. Computing* 14 (1985), pp. 781-798.

[8] D. E. Knuth, *The Art of Computer Programming, Vol. 1, Fundamental Algorithms*, Addison-Wesley, Reading, MA, 1968.

[9] J. Hopcroft and R. E. Tarjan, "Algorithm 447: Efficient algorithms for graph manipulation," *Comm. ACM* 16 (1973), 372-378.

[10] J. Hopcroft and R. E. Tarjan, "Dividing a graph into triconnected components," *SIAM J. Computing* 2:3 (1973).

[11] R. Karp and V. Ramachandran, "Parallel Algorithms for Shared Memory Machines," Tech. Rep. No. CSD 88/408, Dept. of Computer Science, U.C. Berkeley (1988). (To appear in *Handbook of Theoretical Computer Science*, North-Holland.)

[12] J. H. Reif, "A topological approach to dynamic graph connectivity," *Inform. Process. Lett.* 25 (1987), pp. 65-70.

[13] A. Schonhage, "Storage modification machines," *SIAM J. Comput.* 9 (1980), pp. 490-508.

[14] D. D. Sleator and R. E. Tarjan, "A data structure for dynamic trees," *J. Comput. Sys. Sci* 26 (1983), pp. 362-391.

[15] D. D. Sleator and R. E. Tarjan, "Self-adjusting binary search trees," *J. Assoc. Comput. Mach.* 32 (1985), pp. 652-686.

[16] R. E. Tarjan, ''A class of algorithms which require nonlinear time to maintain disjoint sets,'' *J. Comput. Sys. Sci.* 18 (1979), pp. 110-127

[17] _____, ''Amortized computational complexity,'' *SIAM J. Alg. Disc. Meth.* 6 (1985), pp. 306-318.

[18] _____, *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, 1983.

[19] _____, ''Depth first search and linear graph algorithms,'' *SIAM J. Computing* 1 (1972), pp. 146-160.

[20] _____, ''Efficiency of a good but not linear set union algorithm,'' *J. Assoc. Comput. Mach.* 22 (1975), 215-225.

[21] R. E. Tarjan and J. van Leeuwen, ''Worst-case analysis of set union algorithms,'' *J. Assoc. Comput. Mach.* 31 (1984), pp. 245-281.

[22] R. E. Tarjan and U. Vishkin, ''An efficient parallel biconnectivity algorithm,'' *SIAM J. Computing* 14 (1985), pp. 862-874.

[23] J. Westbrook and R. E. Tarjan, ''Amortized analysis of algorithms for set union with backtracking,'' *SIAM J. Computing* 18 (1989) pp 1-11.

[24] A. C. Yao, ''Should tables be sorted?'' *J. Assoc. Comput. Mach.* 28 (1981), 615-628.